ESTRUTURAS DE DADOS E ALGORITMOS

Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 8
LISTAS LIGADAS

8.1. Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de lista ligada;
- Perceber as vantagens da utilização das listas ligadas em relação às tabelas;
- Saber manipular de forma adequada o conceito de lista ligada de forma a implementar/alterar funções que manipulam essas listas ligadas.

8.2. Introdução às listas ligadas

As listas ligadas (*linked list*) são utilizadas para armazenar conjuntos de dados do mesmo tipo. Como vimos anteriormente, podemos também utilizar tabelas para esse efeito. Contudo, uma tabela é uma estrutura sequencial constituída por elementos que ocupam zonas de memória contíguas. O tamanho máximo de uma tabela é fixo, o que implica que mesmo quando vazias estas podem ocupar um grande espaço na memória. As inserções ou remoções de novos valores no meio de uma tabela ordenada implicam normalmente a movimentação de um grande número dos seus elementos.

Quando o número de elementos a armazenar não é conhecido *a priori*, ou quando os elementos necessitam de ser inseridos ou eliminados com frequência, é preferível utilizar listas ligadas. A razão para isso prende-se com o facto de cada elemento de uma lista ligada poder ser armazenado em qualquer lugar da memória sem que exista nenhum tipo de restrição entre elementos consecutivos.

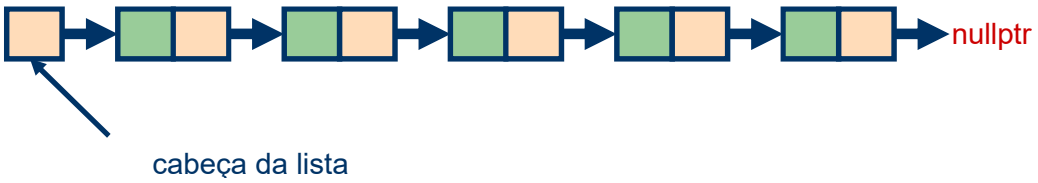
Cada elemento numa lista ligada é designado por **nó**. Numa lista ligada simples cada nó armazena dois tipos de informação:

- O **Dado** a armazenar. Este pode ser de qualquer tipo (*standard* ou definido pelo utilizador).
- O **Ponteiro** (ou ligação) para o próximo nó.

Podemos definir a seguinte classe `CNoLista` para representar um nó de uma lista ligada de inteiros:

```
class CNoLista{
  public:
       int dados;
       CNoLista *proximo;
};
```

Esta classe possui o atributo `dados` que representa a informação a armazenar (neste caso um inteiro) e o atributo `proximo` do tipo ponteiro que permite aceder ao próximo elemento na lista. Uma lista ligada é consequentemente definida como um conjunto de vários nós ligados entre si.



Para definir a lista, apenas necessitamos de armazenar um ponteiro para o primeiro nó da lista. Este ponteiro é normalmente designado por **cabeça da lista**. Como se pode observar, o último nó da lista contém, no atributo `proximo`, um ponteiro vazio (`nullptr`) para indicar o fim da lista.

Podemos representar uma **lista** de inteiros através da definição da seguinte **classe**:

```
class CListaInteiros{
      CNoLista *cabeca;   // ponteiro para 1º nó da lista
  public:
      CListaInteiros(void);
      ~CListaInteiros(void);

      void insereItem(int);
      void insereItemOrdenado(int);
      void apagaItem(int);
      bool procuraItem(int) const;
      void escreveLista(void) const;
      ...
};
```

A classe `CListaInteiros` apenas armazena, como já foi referido, um ponteiro para o primeiro elemento da lista (**atributo** `cabeca`). Para além disso, são definidos alguns **métodos** que permitem manipular a lista. O primeiro dos métodos indicados é o **construtor** que apenas inicializa o membro `cabeca` com o valor `nullptr`, indicando que a lista está vazia inicialmente.

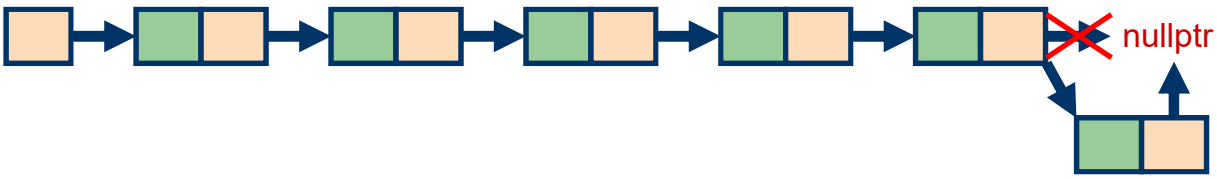
```
CListaInteiros :: CListaInteiros() {
      cabeca = nullptr;
}
```

É importante também definir um **destrutor**, uma vez que antes de se destruir o objeto `CListaInteiros` é necessário libertar a memória associada a cada nó. Para tal, percorre-se a lista utilizando dois ponteiros auxiliares que apontam, em cada instante, para o elemento atual e para o próximo elemento. Desta forma, é possível destruir o elemento atual sem perder uma ligação aos elementos seguintes da lista (apontados por `proximo`).

```
CListaInteiros :: ~CListaInteiros() {
	CNoLista *atual, *proximo;

	atual = cabeca;
	while (atual != nullptr){
		proximo = atual->proximo;
		delete atual;
		atual = proximo;
	}
}
```

Para **inserir** um novo elemento na lista, temos duas possibilidades: ou assumimos que a lista não está ordenada e inserimos o novo elemento no final da lista (ou no início), ou então assumimos que a lista está previamente ordenada e nesse caso introduz-se o novo elemento na sua ordem correta.

Comecemos pelo primeiro caso. A figura a seguir representa a inserção de um novo elemento no final da lista. Em primeiro lugar, é necessário percorrer a lista até se chegar ao último elemento. Depois basta atribuir ao ponteiro do último nó (que tem o valor `nullptr`) o valor do ponteiro para o novo nó.



O algoritmo para inserir um novo elemento no final da lista pode ser descrito da seguinte forma, em pseudocódigo:

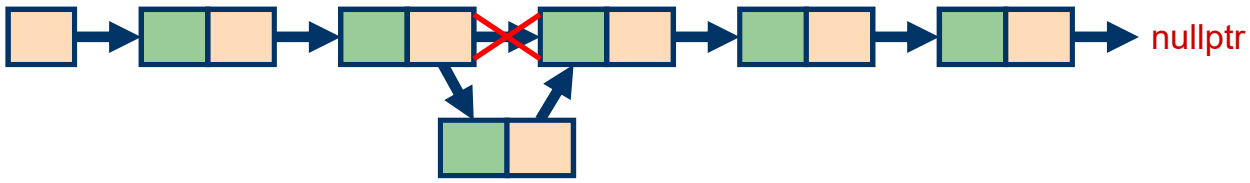
__1__. Criar o novo nó.
__2__. Guardar os dados no novo nó, inicializando o ponteiro «proximo» a nullptr.
__3__. Se a lista estiver vazia
Colocar a cabeça da lista a apontar para o novo nó.
Caso contrário
Percorrer a lista até se atingir o último nó.
Colocar o último nó a apontar para o novo nó.
Fim Se.

A implementação do método `void insereItem(int)`, utilizado para inserir um novo inteiro no fim da lista, é a seguinte:

```
void CListaInteiros :: insereItem(int item) {  // insere no final da lista
	CNoLista *novo=new CNoLista;
	CNoLista *atual;

	novo->dados = item;
	novo->proximo = nullptr;

	if (cabeca == nullptr) cabeca = novo;
	else{
		atual = cabeca;
		while (atual->proximo != nullptr) atual = atual->proximo;
		atual->proximo = novo;
	}
}
```

Dado que a inserção no final da lista é pouco eficiente com listas longas, é preferível, nesse caso, inserir à cabeça, se isso for possível no problema concreto a resolver.

Para inserir um novo elemento respeitando o critério de ordenação previamente definido numa lista ordenada, podemos implementar o método ilustrado na figura seguinte.



Neste caso, e após se ter encontrado a posição correta para a inserção, é necessário alterar os ponteiros do nó anterior para passar a apontar para o novo nó e também o ponteiro do novo nó para passar a apontar para o nó seguinte. Esta regra é alterada se a posição de inserção for no primeiro nó da lista. Neste caso, atualiza-se o ponteiro para a cabeça da lista com o ponteiro para o novo elemento e faz-se o novo elemento apontar para o antigo primeiro nó.

A implementação do algoritmo de inserção ordenada é a seguinte:

```
void CListaInteiros :: insereItemOrdenado(int item) {
	CNoLista *novo = new CNoLista; // aloca espaço para o novo nó
	CNoLista *atual, *anterior;

	novo->dados = item;	// inicializa os dados do novo nó
	novo->proximo = nullptr;

	if (cabeca == nullptr) {  // se lista vazia então este é o primeiro nó
		cabeca = novo;
		return;
	}			// continua na página seguinte...
```

```
	// continuação da página anterior...

	atual = cabeca;
	anterior = nullptr;
	while((atual != nullptr) && (atual->dados<item)) {   // procura a posição de
		anterior = atual;                                // inserção
		atual = atual->proximo;
	}
	if (anterior == nullptr)  // se o ponteiro para o elemento anterior
		cabeca = novo;      // for nullptr então insere no princípio
	else
		anterior->proximo=novo; // caso contrário insere entre dois nós

	novo->proximo = atual; // em qualquer dos casos atualiza o ponteiro
}                              // para o próximo elemento no novo nó.
```

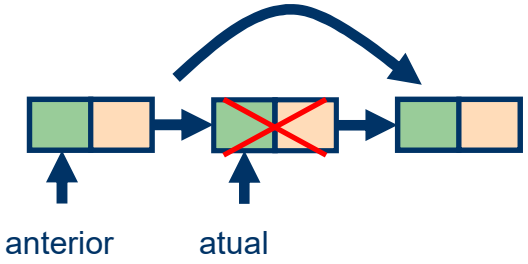
O método seguinte permite a visualização no ecrã dos valores armazenados na lista. Para tal, é necessário percorrer a lista desde o primeiro nó até ao último. Este processo designa-se normalmente por **travessia** da lista (*list traversal*).

```
void CListaInteiros :: escreveLista(void) const {
	CNoLista *atual = cabeca;

	if (cabeca == nullptr) cout << "Lista Vazia..." << endl;
	else {
		while (atual != nullptr) {
			cout << atual->dados << ", "; // tarefa a fazer em cada nó
			atual = atual->proximo;
		}
		cout << "FIM" << endl;
	}
}
```

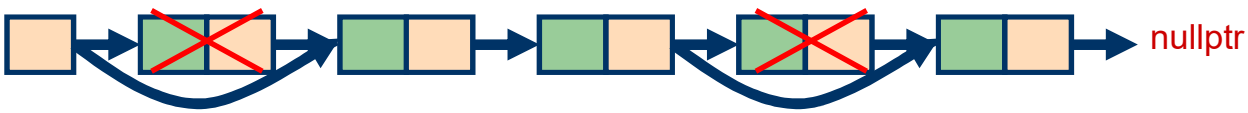
Um outro método, semelhante ao anterior, permite **procurar** um determinado item na lista. Este método devolve um valor lógico que indica se o elemento existe ou não na lista.

```
bool CListaInteiros :: procuraItem(int Item) const {
	CNoLista *atual = cabeca;

	if (cabeca == nullptr) return false;

	while (atual != nullptr) {
		if (atual->dados == item) return true;
		atual = atual->proximo;
	}
	return false;
}
```

O último método da classe `CListaInteiros` descrito aqui como exemplo permite a **eliminação** de um dado item da lista. Para tal é necessário, em primeiro lugar, encontrar o nó que contém o elemento a eliminar. Na procura, utiliza-se mais uma vez dois ponteiros, `atual` e `anterior`, que apontam respetivamente para o elemento atual e para o elemento que o antecede na lista.



Consoante a localização do nó a eliminar, é apenas necessário atualizar o valor do ponteiro guardado no elemento anterior, colocando-o a apontar para o nó posicionado a seguir ao elemento a eliminar. Tal como no caso da inserção ordenada, também aqui é necessário fazer a distinção entre a eliminação de um nó no meio da lista e a eliminação do primeiro nó da lista. A figura seguinte representa de forma esquemática o processo de eliminação de um elemento de uma lista ligada:



O algoritmo pode ser descrito da seguinte forma em pseudocódigo:

__1__. Se a lista não estiver vazia.
    __2__. Encontrar o nó pretendido.
    __3__. Se o elemento a eliminar é o primeiro elemento da lista:
        Colocar a cabeça da lista a apontar para o nó a seguir ao nó a eliminar.
    Caso contrário:
        Colocar o elemento anterior ao nó a eliminar a apontar para o elemento a seguir.
    Fim Se.
    __4__. Libertar a memória associada ao elemento a eliminar.
Fim Se.

A implementação deste método é apresentada a seguir:

```
void CListaInteiros :: apagaItem(int item) {

	CNoLista *atual = cabeca;
	CNoLista *anterior;

	if (cabeca == nullptr) return;   // desnecessário

	anterior = nullptr;
	while (atual != nullptr) {
		if (atual->dados == item) {  // elimina nó
			if (anterior == nullptr)
				cabeca = atual->proximo;
			else
				anterior->proximo = atual->proximo;
			delete atual;
			return;
		}
		anterior = atual;
		atual = atual->proximo;
	}
}
```

8.3. Exercícios sugeridos

Com o conjunto de exercícios apresentados a seguir pretende-se que o aluno consolide os seus conhecimentos sobre listas ligadas. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas. Todos os exercícios assumem a definição da classe `CListaInteiros` apresentada anteriormente. O código fonte da declaração desta classe e da definição de alguns dos seus métodos foi disponibilizado no InforEstudante.

Problema 8.1 – Fácil

Defina o método `int numeroElementosPares(void)` que devolve o número de elementos pares contidos na lista de números inteiros.

Problema 8.2 – Fácil

Defina o método `maioresQue(int numero)` que devolve o número de elementos contidos na lista de números inteiros que sejam maiores do que o número inteiro passado como parâmetro.

Problema 8.3 – Fácil

Considere que a lista ligada está ordenada. Defina o método `existeNumero(int numero)` que verifica se o número inteiro passado por parâmetro existe ou não na lista de números inteiros.

Problema 8.4 – Fácil

Defina o método `int ultimoElemento(void)` que devolve o último elemento contido na lista de números inteiros. No caso de a lista estar vazia, este método deve devolver -1.

Problema 8.5 – Médio

Defina a sobrecarga do operador `[ ]` de modo a devolver o elemento cujo nó ocupa na lista de números inteiros uma dada posição dada pelo parâmetro do operador. O índice zero corresponde ao primeiro elemento da lista. Caso o valor correspondente ao índice não exista na tabela, o operador deve devolver −1.

Problema 8.6 – Médio

Defina o método `void eliminaImpares( )` que elimina todos os números ímpares da lista de números inteiros.

Problema 8.7 – Médio

Defina um novo construtor `CListaInteiros(char *str)` que cria uma lista de inteiros a partir do código ASCII dos carateres que compõem uma *string* passada como parâmetro. Exemplo: se a *string* passada como parâmetro for "ABETO" o construtor deve criar a lista: 65→66→69→84→79.

Problema 8.8 – Médio

Defina a sobrecarga do operador == de forma a permitir que esse operador possa ser utilizado para verificar se duas listas de números inteiros são iguais. Considere que duas listas são iguais quando ambas contêm o mesmo número de elementos.

Problema 8.9 – Difícil

Defina a sobrecarga do operador `+=` para que este permita adicionar à primeira lista de números inteiros os números contidos na segunda, desde que estes ainda não existam na primeira lista.
A segunda lista não deve ser modificada. No caso de a segunda lista estar vazia, o operador não deve fazer nada.

Problema 8.10 – Difícil

Defina o método `void inverteLista(void)` que permite inverter a ordem dos membros da lista de números inteiros sem criar novos nós ou apagar os já existentes.

Problema 8.11 – Difícil

Defina a sobrecarga do operador + para devolver uma lista de números inteiros que representa a fusão de duas listas numa só. No caso de ambas as listas terem itens com o mesmo valor, apenas deve ser inserido um deles na lista devolvida como resultado.

Problema 8.12 – Difícil

Defina a sobrecarga do operador `-=` para eliminar da primeira lista de números inteiros todos os elementos que também pertençam à segunda lista.

Problema 8.13 – Médio **(saiu no teste de frequência de 04/05/2016)**

Considere uma lista ligada `CListaOcorrencias` que permite armazenar o número de ocorrências de cada palavra num dado livro.

```
class CNoLista {
   public:
      char Palavra[25];
      unsigned int Contador;
      CNoLista *Proximo;
};
```

```
class CListaOcorrencias {
      CNoLista *PrimeiraPalavra;
   public:
      CListaOcorrencias (void);
      ~CListaOcorrencias (void);
        ...
      CListaOcorrencias & operator += (char *NovaPalavra);
};
```

a) Implemente um novo método `porOcorrencia()` que permita devolver a palavra contida na lista com o maior número de ocorrências, bem como esse número de ocorrências. No caso de haver mais do que uma palavra com o mesmo número de ocorrências, deverá apenas devolver uma delas.

b) Implemente nesta classe a sobrecarga do operador `+=` de forma a permitir adicionar uma nova palavra à lista de ocorrências (ver protótipo). Se a palavra já existe na lista, incrementa-se apenas o contador. Se a palavra ainda não existe, acrescenta-se a nova palavra no fim da lista e coloca-se o respetivo contador a um.

Problema 8.14 – Médio **(saiu no exame de recurso de 05/07/2016)**

Pretende-se implementar uma aplicação informática para gerir o sistema de apoio a clientes numa determinada empresa. Para isso decidiu-se criar uma lista ligada denominada `CListaHelpdesk` que armazena a informação relativa a todas as perguntas mais frequentes que são feitas pelos clientes. Cada elemento da lista, `CNoLista`, armazena pelo menos a seguinte informação: código da pergunta (5 carateres), resposta à pergunta (300 caracteres) e o número de vezes que a pergunta foi pesquisada (inteiro).

a) Declare as classes `CListaHelpdesk` e `CNoLista` que permitem armazenar a lista das perguntas mais frequentes.

b) Para diminuir o tempo gasto em localizar uma determinada pergunta na lista, de cada vez que é feito um acesso a um nó (para consulta ou alteração de um dos seus campos de informação), este nó é movido para o início da lista. Deste modo, os nós acedidos com maior frequência ficarão localizados no início da lista. Implemente um método `pesquisaLista()` que permita procurar uma pergunta na lista (dado o seu código) e devolver a resposta associada a essa pergunta. Pretende-se também que no caso de a pesquisa ter tido sucesso (o código foi encontrado), se mova o nó correspondente para o início da lista e se incremente o número de vezes que a pergunta foi pesquisada.

Problema 8.15 – Difícil **(adaptado do teste de frequência de 16/06/2017)**

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo construtor que aceite como parâmetros uma tabela de inteiros e a sua dimensão, e que construa uma lista com os números contidos na tabela. O primeiro número da tabela deve ficar na cabeça da lista e o último na cauda. O método não pode chamar outros métodos da classe.

b) Defina um novo método da classe que realize a sobrecarga do operador `<<`. O operador "roda" os nós armazenados na lista `n` (tipo `int`) posições, no sentido da cauda para a cabeça da lista. Por exemplo, se chamar o método para a lista `7, 5, 9, 0, 6, 10` e com `n=2`, a lista fica `9, 0, 6, 10, 7, 5`. O método não faz nada se `n` for menor que 1 ou maior ou igual que o número de nós da lista. O método não pode chamar outros métodos da classe.
Sugestão: Antes de começar a escrever o código, desenhe um esquema para perceber bem a estrutura do problema e desenhar com sucesso um algoritmo para o resolver.

Problema 8.16 – Médio **(adaptado do exame de recurso de 03/07/2017)**

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo construtor que aceita como parâmetros dois inteiros e constrói uma lista com todos os números sequenciais crescentes entre esses dois inteiros (inclusive). O menor valor dos dois parâmetros deve ser guardado no elemento situado na cabeça da lista e o maior na cauda. O método não pode chamar outros métodos da classe.

b) Defina o método constante `bool CListaInteiros::temNumeroImpar(void) const` que verifica se a lista tem um número par de elementos.

c) Defina o método `CListaInteiros& CListaInteiros::operator *= (const CListaInteiros &outra)` que multiplica o 1.º elemento da lista pelo 1.º elemento da lista `outra`, o 2.º elemento da lista pelo 2.º elemento da lista `outra`, e assim sucessivamente. Se a lista estiver vazia, o método não faz nada. Se a lista tiver mais elementos que a lista `outra`, os elementos que não tiverem correspondência com a lista `outra` devem permanecer inalterados.

Problema 8.17 – Médio **(saiu no teste de frequência de 16/05/2018)**

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina o método constante `int CListaInteiros::getEnesimoItem(int)` que devolve o enésimo elemento da lista. Se a lista estiver vazia ou o enésimo elemento não existir, o método deve devolver `INT_MIN`. Por exemplo, `getEnesimoItem(23)` devolve o 23º elemento da lista (se este existir).

b) Defina o método `CListaInteiros& CListaInteiros::operator--(void)` que decrementa cada elemento da lista.